Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 63 (2015) 356 – 365

Procedia Computer Science

The 5th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2015)

Dynamic Healthcare Interface for Patients

Elhadi M. Shakshuki[a]*, Malcolm Reid[a], Tarek R. Sheltami[b]

[a]*Acadia University, 15 University Ave, Wolfville, NS B4P 2R6, Canada*
[b]*King Fahd University of Petroleum and Minerals, Dhahran, 31261, Saudi Arabia*

Abstract

Canadian healthcare is a fundamental part of society. Challenges such as the aging baby boomer generation require the healthcare industry to meet higher demands while using fewer resources. Computer systems designed to record and report physical health properties of an individual person can be used in part to accomplish this task. In this paper, we present the architecture of a hypothetical multi-agent system designed to provide healthcare information about specific patients through continuous monitoring. The resulting data from the system is accessible by the patient to whom it belongs as well as his or her healthcare professional. Furthermore, the proposed system utilizes an adaptive user interface for the purpose of improving the overall experience for users with poor vision or motor skills. Specifically, we focus on the implementation of several of the key components involved in the adaptive user interface: learning component and the user model. To demonstrate the feasibility of the implementation two scenarios are provided. We conclude with several possible future directions for this research.

Peer-review under responsibility of the Program Chairs

Keywords: Multi-Agent Systems; Healthcare Technology; Adaptive User Interface; Reinforcement Learning

1. Introduction

Healthcare is a fundamental part of Canadian culture. However, the aging baby boomer generation presents a growing challenge to Canada's existing health infrastructure. In Canada, the number of citizens reaching the age of retirement and seniority is growing faster than the rate of citizens entering the workforce to provide the health related services[1, 2]. When citizens reach seniority they tend to require more frequent checkups and health services, ultimately placing a bigger resource drain on the healthcare industry.

* Corresponding author. Tel.: 1-902-585-1524
E-mail address: elhadi.shakshuki@acadiau.ca

Peer-review under responsibility of the Program Chairs
doi:10.1016/j.procs.2015.08.354

As demand for healthcare services increases new solutions to overcome the fiscal and logistical challenges are required in order to continue providing the quality of healthcare expected by the population.

Advancements in computer science and engineering such as multi-agent systems (MAS) and miniature low power processors are creating one possible solution for providing healthcare and related services[3,4]. Utilizing a MAS design paradigm allows systems in which software agents autonomously act on behalf of their human users. New technologies like this provide the opportunity for creating computer systems designed to record and report physical health properties of an individual person, which in turn can be used to reduce the resource and person cost required of providing healthcare services.

In this paper, we present an architecture for a multi-agent system designed to gather statistical health information about patients using continuous monitoring from sensors in and on the human body. Patient can access his or her information in the system using a handheld mobile device. Nurses, doctors, and other healthcare professionals related to the user in question will also have access to that data through a similar interface. Furthermore, the proposed system utilizes an adaptive graphical user interface for the purpose of improving the overall experience for users with poor vision or motor skills.

The main focus of this paper is to introduce the system architecture required for the adaptive user interface of the proposed system. Two required agent components for an adaptive user interface are implemented and presented in section 4, followed by two scenarios demonstrating the feasibility of the adaptive user interface. Lastly section 5 concludes with some observations and future directions for this research.

2. Related Work

This section focuses on evaluating and comparing existing research works related to architectures of adaptive UI. We briefly describe the architectures and argue their strengths and shortcomings, and conclude by comparing them.

A *3-layer architecture* was presented for developing adaptive smart environment user interfaces[5]. Due to the ubiquitous nature of its target applications, this architecture only supports direct adaptations. Information is read from sensors, and the environment context pillar is targeted and as such, multiple data sources are not supported. The architecture uses a modeling approach based on generative runtime models, which could be less flexible than interpreted runtime models for performing advanced adaptations[6]. Furthermore, the work does not specify whether the architecture is meant to support all levels of abstraction. The architecture does not support user feedback but refers to the work of Brdiczka et al. that does not offer an architecture, but uses user-feedback for refining initial situation models at runtime in order to improve the reliability of detected situations[7].

CAMELEON-RT is a reference architecture model for distributed, migratable, and plastic user interfaces within interactive spaces[8]. This architecture targets all context-of-use pillars (that is: user, platform, and environment), and can be considered general-purpose due to its implementation neutrality[6]. The architecture provides a good conceptual representation of the extensibility of adaptive behaviour through the use of open-adaptive components, which allow new adaptive behaviour to be added at runtime[9]. Both direct and indirect adaptations could in theory be implemented using these components. The CAMELEON framework supports all levels of abstraction. The architecture depicts observes that collect data on the system, user, platform, and environment, and feed it to a situation synthesizer thereby supporting multiple data sources[6].

CEDAR is a reference architecture for stakeholders interested in developing adaptive enterprise application UIs based on an interpreted runtime model-driven approach[10]. The architecture follows the levels of abstraction suggested by CAMELEON for representing its UI models[6]. It supports both direct and indirect adaptation and the extensibility of its adaptive behaviour, which is stored in a relational database. CEDAR presents components for supporting trade-off analysis and user feedback on the UI adaptations. Furthermore, the architecture was evaluated by integrating it into an existing enterprise application called OFBiz[6]. Lastly, the architecture also introduced a basic crowdsourcing approach for empowering end-users to participate in the UI adaption process[6].

FAME is an architecture targeting adaptive multimodal UIs using a set of context models in combination with user inputs[11]. It only targets modality adaptation and is therefore not meant to be a general-purpose reference for adapting other UI characteristics. The adopted approach allows designer input on the character user interface; hence, providing good control over the UI. Adaptive behaviour can be extended using device changes, environmental changes, and user inputs that feed into related models[6]. According to Akiki et al. the combination of the multiple data sources and the adaptive behaviour matrices should be able to support both direct and indirect adaptations[6].

Malai is an architectural model for interactive systems and forms a basis for a technique that uses aspect-oriented modeling for adapting user interfaces[12,13]. The extensibility of adaptive behaviour is poor since multiple presentations have to be defined at design-time by the developer, to be later switched at runtime[6]. Although Malai supports multiple levels of abstraction, the modeling approach relies on generating code (such as Swing, .NET, etc.) to represent the UI. Furthermore, it does not describe multiple sources for acquiring adaptive behaviour data. In theory, both direct and indirect adaptations can be supported[6]. Malai allows developers to define feedback that would help users to understand the state of the interactive system, but the user cannot provide feedback on the adaptations (i.e., reverse an unwanted adaptation).

After analyzing the works reviewed in this section, it became clear that most of the architectures did not address several key criteria. For example, only CEDAR presents components for managing trade-off analysis and user feedback[6]. Additionally, despite the importance of integration in existing software systems that are in a mature development stage all of the evaluations, except CEDAR, were conducted by building new prototypes. Furthermore, empowering new design participants was only partially addressed by CEDAR, while the other architectures did not incorporate any components for supporting this feature[6]. Akiki et al. present a visual evaluation and comparison of the architectures discussed in this section in Figure 1. Our work is an attempt to combine and improve upon the techniques discussed in this section to create an architecture for a healthcare application where direct and indirect UI adaptations are possible utilizing the three pillars of context-of-use: user, platform, and environment.

Legend
● Completely fulfills
◖ Partially fulfills
○ Does not fulfill
○ Not specified

	Direct and indirect adaptation	Empowering new design participants	Extensibility of adaptive behavior	Integrating in existing systems	Levels of abstraction	Modeling approach	Multiple data sources	Trade-off analysis	User feedback on the adapted UI
3-Layer Architecture	◖	○	○	○	○	◖	◖	○	○
CAMELEON-RT	●	○	●	○	●	○	●	○	○
CEDAR	●	◖	●	●	●	●	●	●	●
FAME	●	○	●	○	◖	○	●	○	○
MALAI	●	○	◖	○	●	◖	◖	○	○

Figure 1: Feature comparison of existing AUI architectures[6].

3. System Architecture

The healthcare environment in this study consists of healthcare professionals, patients, and software agents. Patient users (PUs) are the people being monitored by the proposed system architecture. Healthcare professional users (HUs) consist of doctors, nurses, and technical support personnel.

Software agents in the environment are: user agent (UA) and resource agent (RA). Each human user is assigned a UA. There will be many user agents in the system, and one resource agent. The RA is responsible for writing historical data to a central server repository. This allows for the later retrieval and analysis of important patient data. An additional role of RA is to act as a security authority and authenticate any patient data requests.

User agents can be operated by either a healthcare professional or a patient, and must behave accordingly. Patients will be wearing a BASN for monitoring health data, which will act as sensor input for the user agent representing them. A patient can use his or her coordinating device to view real-time as well as historical health data belonging to them. Healthcare professionals will have a UA representing them and their respective authority. For example a doctor's UA would have access to patient health data, both historical and real time. Figure 2 shows a visualization of the system architecture. The system architecture consists of two types of agents: user agent and resource agent. User agents have three primary goals: (1) adapting the UI to improve user experience, (2) managing health data, and (3) responding to healthcare professional user agent requests for health data. The resource agent has two primary goals: (1) authenticating information requests between user agents, and (2) archiving patient health data for long-term storage.

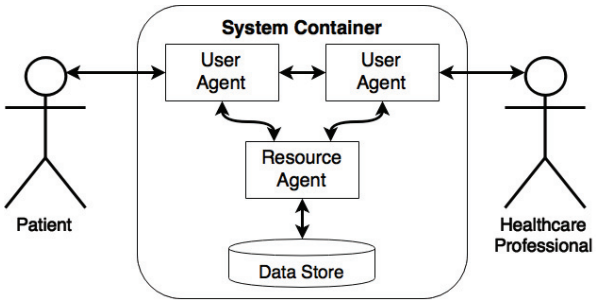

Figure 2: System architecture overview.

Our focus in this paper is on the user agent with the capability of enhancing user experience using reinforcement learning. The UA has the following components: sensor, communication, learning and user model, as shown in Figure 3.

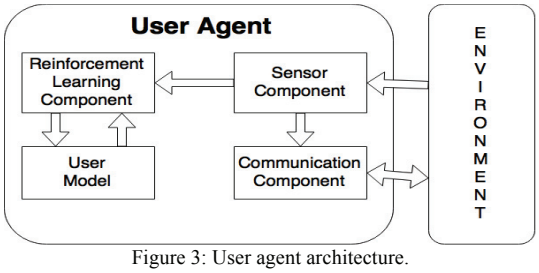

Figure 3: User agent architecture.

The agent's sensor component is responsible for receiving health metrics from the environment via the coordinating device. Once received, health data and any relative agent communications are then passed on to the communication and learning components for further processing. The agent's communication component is responsible for two tasks: 1) transmitting data to the resource agent for historical recordkeeping, and 2) responding to requests by doctor operated user agents. The user model component contains data relating to the usage habits of the human user, both their actions as well as errors. This model is initiated to a zero sum default state and is modified over time by the learning component as the agent attempts to learn the behavior patterns of the user. Lastly, the following section provides a detailed description of the reinforcement learning component.

3.1. Reinforcement Learning Component

The reinforcement learning component (RLC) is responsible for learning the behaviour of a user by tracking his or her historical actions as well as errors in interface use. The RLC can be broken down into three sub-components: 1) learning, 2) evaluating, and 3) adapting. See Figure 4 for an illustration of how the sub-components relate and interact with each other and the system environment around them.

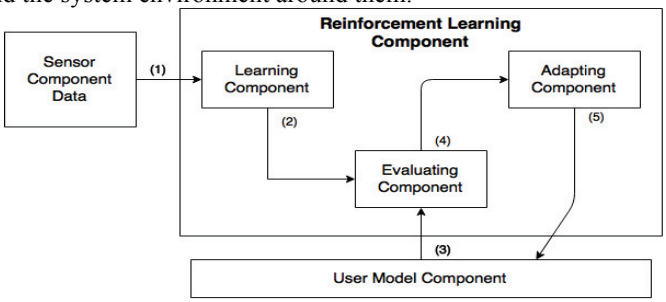

Figure 4: Reinforcement learning component.

Figure 4 also shows the steps as information flows through the RLC. First data enters the RLC from the agent's sensor component. This data may consist of user action choices, such as viewing a health metric, as well as

information about how the user interacted with the system interface. The LC parses the information received into two categories: action, and behaviour. The action group consists of user choices and preferences, while the behavior group consists of statistical interactions with the interface, such as coordinates of touch-screen clicks or ambient light levels. User action might not be included with the usage data; an example of this would be a failed attempt to tap a button. Once actions and behaviours are identified they are passed to the Evaluating Component (EC).

The evaluating component is responsible for pulling relative information, with respect to the received usage data, from the agent's perceived concept of the user, which is stored in the user model component. The EC compares the new usage actions with the existing policies. If there is no data about that action in the policy already, then nothing happens and the data is passed on to the next component. However, if there is information in the agent's user model then the EC must compare it with the new usage data and determine the differences. Next, data is passed to the Adapting Component (AC). The AC receives the new usage data as well as the related data from the user model for that action. The AC must determine what actions need to be taken and update the user model accordingly.

4. Implementation

The reinforcement learning and user model components from our proposed system are implemented in java as a proof of concept. The Java Agent Development Framework (JADE) was chosen as a starting platform. The framework consists of the required infrastructure for deploying MAS as well as basic starting classes for agent development[14].

4.1. User Agent Implementation

The user agent is implemented using an agent-oriented design. Each component in Figure 3 is implemented as its own object within the UA: 1) learning component, 2) sensor component, 3) communication component, and 4) the user model component. The primary focus of this paper is on the reinforcement learning component and the user model component, which are both described in more detail in the following sections. Although previously defined as a handheld smart device, the system interface was implemented as a Java GUI. The complexities and challenges of developing an AUI in iOS or Android will be explored in future work.

4.2. Reinforcement Learning Component

Let us assume the interface has an ability to record user behaviour. Which is defined as actions and interactions as a user interacts with objects in the interface. This behaviour information is received by the learning component, as shown in Figure 4. The RLC is implemented using object oriented design techniques whereby each subcomponent is implemented as its own class: 1) learning component, 2) evaluating component, and 3) adapting component.

4.2.1. Learning Component

The learning component (LC) receives action choices and physical interactions made by the user as input. The main goal of the LC is to separate this data into actions and behaviours.

LearningComponent
+ actions: String + behaviours: String + usageData: String
+ separate(usageData, actions, behaviours): void

Figure 5: class diagram for the learning component object.

Figure 5 shows a UML markup of the Learning Component class. There is a field for the current usage data as received from the interface, as well as fields for the actions and behaviours associated with the usage data. Initially, these fields are empty. The method *separate* is where the classification of action and behaviour takes place. After execution, the arrays of actions and behaviours (see Table 1) are populated with data from the current usage scenario. Once completed, the RLC passes the actions and behaviours on to the evaluating component.

Table 1: pseudo code for the learning component separate function.

SEPARATE(String usageData, String Action, String Behaviour) : null

```
While usageData still has data to read
    read next data string from usageData
    if next data is an action
        add next data to Action
    else
        add next data to Behaviour
End
```

4.2.2. Evaluating Component

The Evaluating Component (EC) receives two types of input from the LC. They are 1) user actions, or 2) user behaviour. Recall, actions correlate to user choices within the application, and user behaviour relates to the physical interactions between the user and interface. The goal of the EC is to retrieve historical information about either user actions or behaviours.

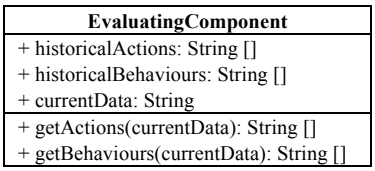

Figure 6: class diagram for the evaluating component.

Figure 6 shows a UML diagram of the Evaluating Component class. EC contains three fields for actions: 1) the current type of action or behaviour being evaluated, 2) historical actions initialized to an empty array, and 3) historical behaviours also initialized to an empty array. Table 2 shows the method *getActions,* which takes the current action or behaviour being evaluated as input, and returns an array of related actions from the user model. In the case that no related actions exist in the User Model, then the array is empty. Similarly, Table 3 shows the method *getBehaviours* that takes the current action or behaviour to be evaluated as input and returns an array of related historical behaviours. In the case where no historical behaviours related to the action or behaviour being evaluated exists, then an empty array is returned. Once historical actions and behaviours have been retrieved from the user model the EC passes the historical data as well as the current data on to the adapting component.

Table 2: pseudo code for the evaluating component getActions function.

```
GET_ACTIONS(String currentData) : String
Open User Model for reading
result is empty
For each action with the same action id as currentData in User Model,
    add it to result
Return result
```

Table 3: pseudo code for the evaluating component getBehaviours function.

```
GET_BEHAVIOURS(String currentData) : String
Open User Model for reading
result is empty
For each behaviour with the same action id as currentData in User Model,
    add it to result
Return result
```

4.2.3. Adapting Component

The adapting component receives three inputs: 1) current action or behavior being evaluated, 2) historical relevant actions, and 3) historically relevant behaviours. It is possible that no historical data is received, when no record of the action or behaviour exists in the user model. The primary goal of the AC is to determine if the current policy for a specific action or behaviour is optimal, given the historical and current usage data.

Figure 7 shows a UML diagram for the Adapting Component (AC). Three fields represent the current action or behaviour being evaluated, the related historical actions, and behaviours found by the EC. The adapting component determines if the current policy is optimal by the functions (1) *compareActions* and (2) *compareBehaviours*. The first function accepts one parameter: the action or behaviour in question. Each related historical action is compared to the parameter. Iterating through all the related actions, a running fitness score is calculated. If this score is negative then adaption is required, on the other hand if the score is positive then no adaptation is needed.

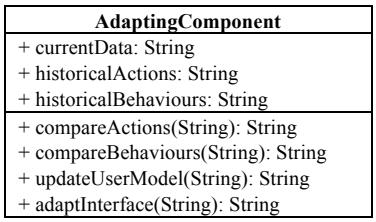

Figure 7: class diagram for the Adapting Component object.

The second function compares the current action or behaviour to the known historically related behaviours, shown in Table 4. Lastly, and perhaps most importantly, if the fitness score determined for the current action or behaviour is negative – meaning an adaption is required – then the *updateUserModel* function shown in table 5 is called. This function performs the task of updating the User Model component with the most current historical action or behaviour being evaluated. Furthermore, the function *adaptInterface* is responsible for updating the GUI components, if required, to match the new policy.

Table 4: pseudo code for the compareActions function.

```
COMPARE_ACTIONS(String action) : String
For each action x in historicalActions:
    if action > x
        fitnessScore++;
    else
        fitnessScore--;
End
Return fitnessScore
```

Table 5: pseudo code for the compareBehaviours, updateUserModel, and adaptInterface functions.

```
COMPARE_BEHAVIOURS(String behaviour) : String
For each action x in historicalBehaviours:
    if action > x
        fitnessScore++;
    else
        fitnessScore--;
End
Return fitnessScore
```

```
UPDATE_USER_MODEL(String data) : null
Open User Model for writing
if data is an action
    update USER_ACTIONS table
else
    update USER_BEHAVIOURS table
```

```
ADAPT_INTERFACE(String adaptation) : null
While adaptation is not empty
    read interface component name
    read adaptation parameters
Modify graphical user interface component as specified
```

4.3. User Model

The user model (UM) is a collection of actions and behaviour describing how a user interacts with the system. A natural conclusion is to use a database to store the data for easy retrieval later. Every time new usage data is received it is compared with the known existing behaviour patterns in the user model. Without it, the interface would not be able to adapt in a meaningful way to each user. Implementation of the UM required a database with a light footprint ultimately, we chose SQLITE. The UM database contains two tables: 1) USER_ACTIONS, and 2) USER_BEHAVIOUR.

The USER_ACTIONS table contains historical actions. These actions are event relative, such as: opening a new page in the interface, requesting a specific type of data, or changing display or setting options. Each action correlates to an event the user performed within the application. The columns of the table consist of: a timestamp as the primary key, and a unique action ID performed. The purpose of this table is to track user actions in order to suggest most frequent actions, in an attempt to improve user experience. The data definition language (DDL) for the USER_ACTION table is: CREATE TABLE USER_ACTIONS (timestamp TIMESTAMP PRIMARY KEY, action_id INT NOT NULL).

The USER_BEHAVIOURS table contains historical usage data of the interactions between the human user and the interface. The columns of the table are: a timestamp as the primary key, the type of action performed, as well as an optional unique action ID associated with the behaviour. If the behaviour performs some action, there would be a reference to that action in the table. The DDL for the USER_BEHAVIOURS table is: CREATE TABLE USER_BEHAVIOURS (timestamp TIMESTAMP PRIMARY KEY, action_type INTEGER NOT NULL, associated_action_id INTEGER).

4.4. Scenario I

The first scenario describes the observed data and subsequent changes in the proposed system when a patient with poor motor skills has trouble clicking a button while operating the interface. Lets assume that the patient in question suffers from hand tremors and wishes to view his cardiograph data (heart rate) for the previous 24 hours, given the data exists within the system and it is 10:30am. Lastly, lets assume that the patient is operating a touch screen device and looking at the screen to request cardiograph data in the proposed system interface. See Figure 8 (A) for a mock-up of this screen.

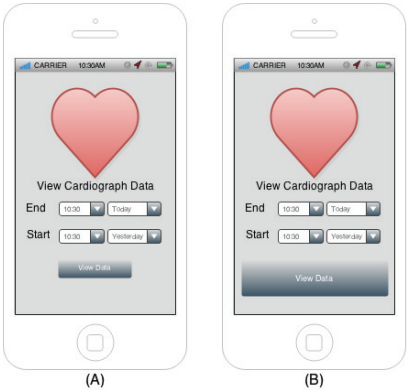

Figure 8: interface screen requesting cardiograph data before (A) and after (B) adapting to the user in scenario I.

The patient interacts with the interface by clicking six times within a fixed margin of the button without touching the button. The fixed margin is determined by each interface component, this scenario assumes the margin is 100 pixels around the button in all directions. On the seventh click, the patient correctly hits the button. At this point, the interface responds and displays the requested information on a new screen. Internally, as soon as the first unsuccessful click is observed by the interface the action data is sent to the user agent's reinforcement learning component. Interface interactions have the form of <*timestamp, description, optional associated action*>. The first six click attempts will have no associated action; however, they do have timestamps and descriptions associated with them. The timestamps are automatically generated as actions are observed, and each description consists of a string of data describing the interface component, and the action information. In this scenario, the first six actions appear as:

<10:30:09, CLICK;BUTTON05;UNSUCCESSFUL;94px;NW,>

<10:30:10, CLICK;BUTTON05;UNSUCCESSFUL;30px;W,>

<10:30:11, CLICK;BUTTON05;UNSUCCESSFUL;50px;E,>
<10:30:14, CLICK;BUTTON05;UNSUCCESSFUL;12px;S,>
<10:30:15, CLICK;BUTTON05;UNSUCCESSFUL;25px;W,>
<10:30:16, CLICK;BUTTON05;UNSUCCESSFUL;9px;E,>

Each action description consists of the action performed and the related user interface element. Furthermore, the action description denotes each click attempt as unsuccessful and provides a description from the interface where the click was located.

The seventh click produces the following action description:

<10:30:17, CLICK;BUTTON05;SUCCESSFUL, VIEW_CARTIOGRAPH_DATA>

The learning component (LC) identifies the six observed actions as behaviours. Next, the EC searches the user model for related historical data, such as previous attempts at clicking *BUTTON05*. Lastly, the AC receives each behaviour action and updates the user model by adding behaviour entries into the database.

The LC identifies the seventh successful click as an action. The EC searches the user model for related historical data, and finds the related data of the previous six click attempts. The AC uses the current action and historical data to determine what course of action to take. In this case, the interface element in question is a button, and the related historical data is several unsuccessful attempts to use the button. Therefore, action course chosen is to increase the size of the button in the GUI, as shown in Figure 8 (B). Finally, the AC updates the user model actions table by inserting the latest action.

4.5. Scenario II

This scenario describes the observed data and subsequent changes to the GUI in our proposed system when a patient with poor vision has trouble operating the interface. Poor vision may include, but is not limited to: blurred vision, colour blindness, and contrast sensitivity. We assume the application contains a setting for visual adaptation (VA), which must be enabled before the interface will adapt. With VA enabled, an acknowledgement button is added to every screen in the interface. The purpose of the button is to let the agent know when the user is able to see the screen clearly. Until the user clicks the acknowledgement button, the interface contrast and colors will slowly change. Over time, the agent will learn the optimal vision settings for the user, and the button can be hidden.

Assume that the patient suffers from contrast sensitivity, and the visual adaptation setting has been enabled within the interface. Suppose the patient in question wishes to view her cardiograph data (heart rate) for the previous 2 hours, given the data exists within the system and it is 1:30pm. Lastly, lets assume that the patient is looking at the screen to request cardiograph data in the proposed system interface. If no previous VA has happened, the screen will be identical to that of Figure 7. The interface will adjust the contrast and brightness of the screen based on the ambient level of light and user preferences. We assume no previous learning has happened, so no user preference is established. Furthermore, the interface contains an acknowledgement button, as shown in Figure 9 (A).

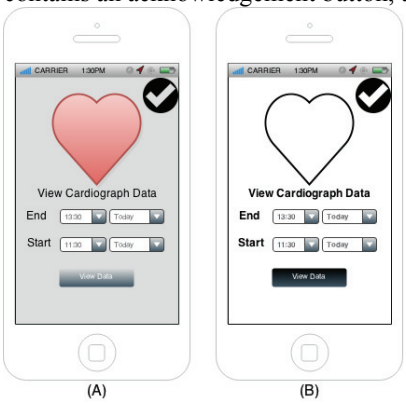

Figure 9: interface screen requesting cardiograph data interface before (A) and after (B) visual adaptation.

Until the RLC observes an interaction with the acknowledgement button, the interface adjusts display settings. Adjustments consist of permutations screen brightness, contrast and color saturation levels. Adjustments are cycled through every 0.5 seconds until a suitable screen setting is acknowledged.

When a suitable setting is found, the RLC halts interface adjustments and records the screen and ambient settings in the user model. This and subsequent observed adjustment settings become the user's default preference. Figure 9 (B) shows an example interface adjustment for this scenario.

5. Conclusion

We presented an architecture for a MAS designed to track and report user health data through continuous monitoring of patients, with the capability of adapting the GUI to individual patient needs. Implementation of the reinforcement learning component and the user model of the proposed system was described. Furthermore, the functionality of the adaptive user interface was demonstrated in two scenarios.

In our future plans we hope to investigate and overcome the challenges of applying reinforcement learning techniques to adaptive user interface design. We intend to investigate how the Markov property behaves with the UI state and human user interaction. Ultimately, we hope to develop an approach for UI adaptation that is independent of the past and current states.

Acknowledgements

The authors would like to thank the research and graduate office at Acadia University for their support.

References

1. CBC News. (2012, May) Canada has higher proportion of seniors than ever before. [Online]. http://www.cbc.ca/news/canada/canada-has-higher-proportion-of-seniors-than-ever-before-1.1151526.
2. CBC News. (2015, January) Why doc? Solution to doctor shortage may be more than money. [Online]. http://www.cbc.ca/news/canada/newfoundland-labrador/why-doc-solution-to-doctor-shortage-may-be-more-than-money-1.2929966.
3. B. H. Calhoun et al., "Body Sensor Networks: A Holistic Approach From Silicon to Users," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 91-106, August 2011.
4. Ashraf Darwish and Aboul Ella Hassanien, "Wearable and Implantable Wireless Sensor Network Solutions for Healthcare Monitoring," *Sensors*, vol. 11, no. 6, pp. 5561-5595, May 2011.
5. G. Lehmann, A. Rieger, M. Blumendorf, and S. Albayrak, "A 3-layer architecture for smart environment models," *Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications.* Mannheim, Germany: IEEE, pp. 636-641. 2010.
6. Pierre A Akiki, Arosha K Bandara, and Yijun Yu, "Adaptive model-driven user interface development systems," *ACM Computing Surveys*, vol. 47, no. 1, pp. In-press, March 2014.
7. O. Brdiczka, J. L. Crowley, and P. Reignier, "Learning situation models for providing context aware services," Springer, 2007.
8. L. Balme, R. Demeure, N. Barralon, J. Coutaz, G. Calvary, and U. J. Fourier, "Cameleon-RT: A software architecture reference model for distributed, migratable, and plastic user interfaces," *Proceedings of the 2nd European Symposium on Ambient Intelligence*. Eindhoven, The Netherlands: Springer, pp. 291-302. 2004.
9. P. Oreizy, et al., "An architecture-based approach to self-adaptive software," *Intelligent Systems and Their Applications*, IEEE, vol. 14, no. 3, pp. 54-62, 1999.
10. P. A. Akiki, A. K. Bandara, Y. Yu, "Using interpreted runtime models for devising adaptive user interfaces of enterprise applications," in *Proceedings of the 14th International Conference on Enterprise Information Systems.* Wroclaw, Poland: SciTePress, pp. 72-77. 2012.
11. C. Duarte and L. Carrico, "A conceptual framework for developing adaptive multimodal applications," in *Proceedings of the 11th International Conference on User Interfaces*, Sydney, Australia: ACM, pp. 132-139. 2006.
12. A. Blouin and O. Beaudoux, "Improvising modularity and usability of interactive systems with Malai," in *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. New York, USA: ACM, pp. 115-124. 2010.
13. A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J. M. Jézéquel, "Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation," in *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems.* Pisa, Italy: ACM, pp. 85-94. 2011.
14. Telecom Italia SpA. (2015, January) Java Agent Development Framework. [Online]. http://jade.tilab.com/.